

# WinFS Storage Architecture

Nigel Ellis  
Development Manager  
WinFS Base

03/02/2004



## WinFS

# WinFS ECS Talks

Overview

Data Model and Programming Model

API

**Architecture and Implementation**

Schema

Rules Engine

File System Integration

Sync

Quentin Clark

Anil Nori

Mike Deem ← **rescheduled**

**Nigel Ellis**

J. Patrick Thompson

Praveen Seshadri

Sanjay Anand

Lev Novik



# Overview

## Covered:

- Basic WinFS architecture
- Deep drilldown into the data model, storage API and implementation
- Details of database features and usage
- Schema development and deployment
- Operations: create-find-update-delete

## Not Covered:

- Sync, Rules or Utilities
  - Deep details of every WinFS component
- 
- More detailed talks on each component in future ECS talks



# What is WinFS?

- WinFS is a rich information store which ships as part of Longhorn
- Data is organized around the concepts of Items, Extensions and Relationships
- This organization is performed in the context of the WinFS data model



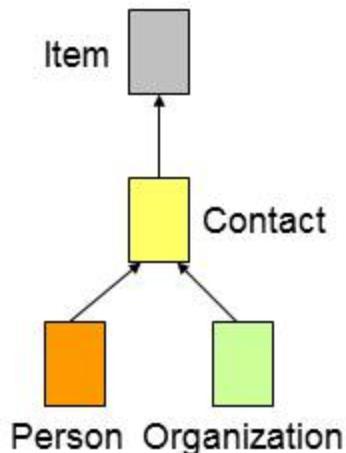
# WinFS Data Model

- The WinFS Data Model describes
  - the shape of the data stored in WinFS
  - the constraints on the data
  - associations between data
- WinFS world is comprised of items, relationships and extensions
- Items are the primary objects that applications work on
- Items can be associated with other items via relationships
- Items can be extended with extensions (or subclass)

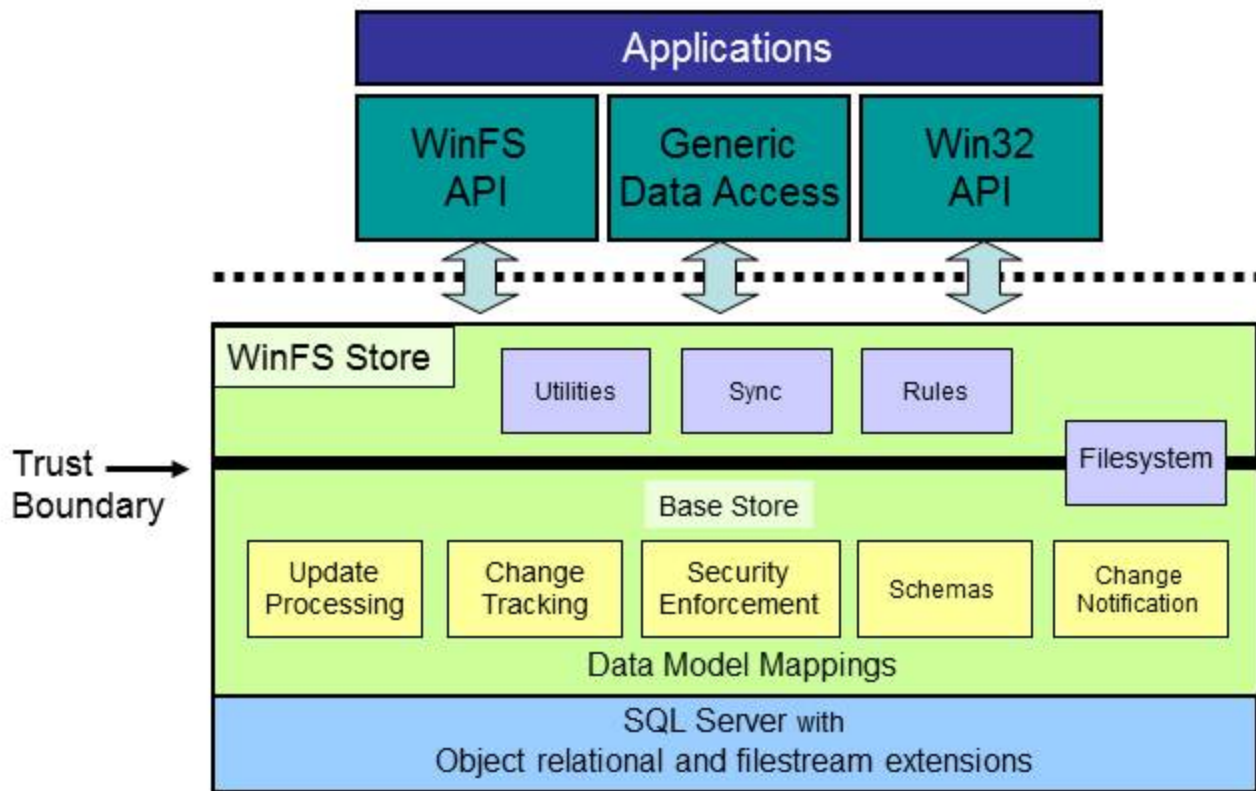


# WinFS Type Example

```
using Contact = System.Storage.Contact;  
using Core = System.Storage.Core;  
using Base = System.Storage;  
  
type Contact.Address : Base.NestedType {  
    string Street;  
    string City;  
    string Zip;  
    ...  
}  
  
type Contact.Person : Core.Contact {  
    datetime BirthDate;  
    binary[] Picture;  
    Address BirthAddress;  
    MultiSet<Address> Addresses;  
    ...  
}  
  
type Contact.Organization : Core.Contact {  
    string OrganizationName;  
    ...  
}
```



# WinFS Layers





# SQL OR Extensions

- WinFS leverages object relational extensions in SQL
- Importing of CLR types into SQL type system
  - Supports single inheritance
  - Type substitution
  - Collections (ordered and un-ordered)
- Objects can be larger than 8K (internal DB page size)
- Lazy materialization
  - Delayed access to object LOB properties
  - Allows you to fetch the Person without the Picture
- Optimizer extensions
  - Indexes on structure paths – Person.Address.City
  - Indexes on nested collections – Person.Offices.Bld
  - Stats on type hierarchy, structured paths, and nested collections





## SQL OR Extensions - Smart Serialization

- SQLCLR types leverage a custom serialization library (SL)
- Efficient access to properties of embedded objects
  - Avoids object construction or method invocations for simple property getters and setters
  - Property and field access translates to compiled field accessors
- New structured serialization format
  - Understands inheritance, embedded types, collection types
  - Internal to SQL
- Provides "record like" performance for accessing object properties



# SQL OR Extensions - Smart Serialization

- Example:

```
SELECT FirstName, LastName, ...  
FROM [System.Storage.Contact.Store].Contact c  
WHERE BirthAddress.City = 'Seattle'
```

- Fetching HomeAddress.City does not require the materialization of the Address object
- Properties retrieved by directly "cracking" the serialized form



## SQL OR Extensions - Collections

- SQL supports a generic collection type `MULTISET<T>`
- Properties can be declared using collections
- Treated from SQL as "nested table"
- Queryable using `UNNEST` table valued function

```
SELECT c.FirstName, c.LastName,  
       A.addr.City, A.addr.Zip  
FROM [System.Storage.Contact.Store].Contact c  
CROSS APPLY UNNEST(c.Addresses) AS A(addr)
```

- Current investigating replacing the `MultiSet` collection with `IList<T>` (to support ordering)



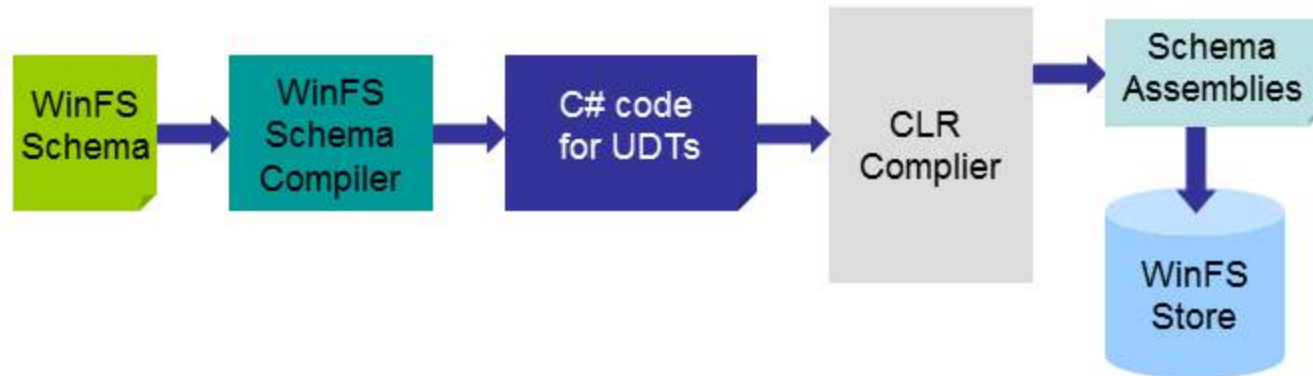
# Schemas

- A WinFS schema defines Item, Extension and Relationship type definitions
- It also defines:
  - Enumerations
  - Views (pre-defined queries)
  - Indexes
- A schema identifies a namespace (CLR)
- The namespace is used to partition types within the WinFS Store and CLR namespace
- The System.Storage namespace defines the root types Item, Extension and Relationship



# Schema compilation process

- WinFS schemas are defined using the WinFS schema language (XML grammar)
- The schema compiler generates C# code from the WinFS Schema
- The C# source files are compiled into assemblies
- Two assemblies are generated for each schema – one client and one storage (.Store) assembly
- The Store assembly is installed into a WinFS store
  - WinFS types are registered as user defined types
  - Views and other database objects are created automatically to provide search and storage for the types
  - Clients of the Store are never allowed to issue direct updates against the Store tables/views/etc. They must use the Base API





# Schema Type Mapping

- Each WinFS Store is backed by a database
- Each storage schema is installed into a Store:

- A SQL namespace is created

```
CREATE SCHEMA [System.Storage.Contact.Store]
```

- The schema assembly is registered

```
CREATE ASSEMBLY [System.Storage.Contact.Store]  
FROM '..\System.Storage.Contact.Store.dll'
```

- Each type in the schema is registered as a SQL UDT  
Sub-types are registered under their parent type

```
CREATE TYPE [System.Storage.Contact.Store].Contact  
EXTERNAL NAME System.Storage.Contact.Store::Contact
```

```
CREATE TYPE [System.Storage.Contact.Store].Organization  
EXTERNAL NAME System.Storage.Contact.Store::Organization
```





# Table Storage

- Item, Extension and Relationship types are stored in a single storage table for each type hierarchy

```
CREATE TABLE [System.Storage.Store].[Table!Item]
  (_Item [System.Storage.Store].Item, ...)

CREATE TABLE [System.Storage.Store].[Table!Extension]
  (_Extension [System.Storage.Store].Extension, ...)

CREATE TABLE [System.Storage.Store].[Table!Relationship]
  (_Relationship [System.Storage.Store].Relationship, ...)
```

- Type instances are stored in object columns `_Item`, `_Extension` and `_Relationship` using type substitutability.
  - For example, Document objects are stored in the column `_Item` (of type `Item`)



# Type Search Views

- Item, Extension and Relationship type provides a “master” search view which projects data directly from the storage item/extension/relationship tables

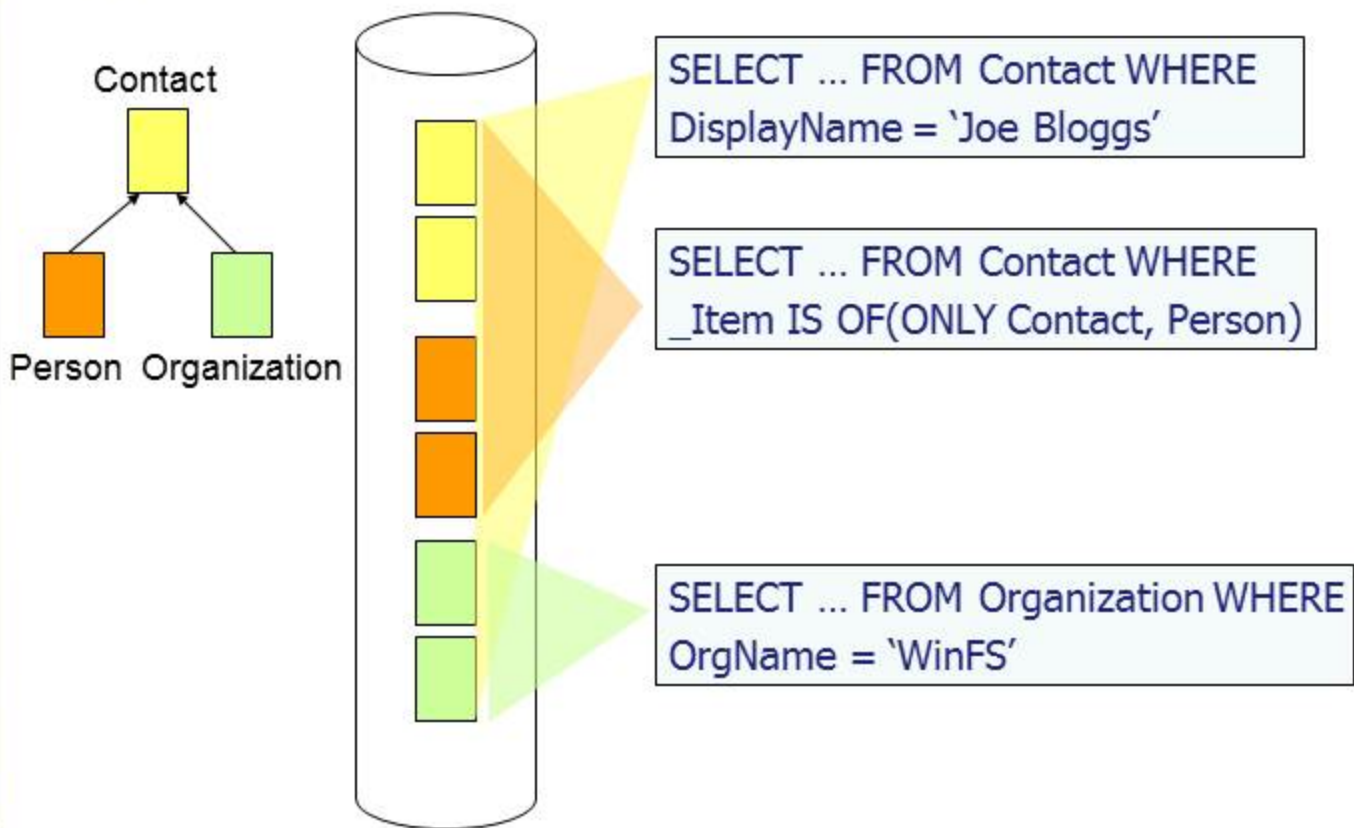
```
CREATE VIEW [System.Storage.Store].[Master!Item] AS
  SELECT ... FROM [System.Storage.Store].[Table!Item]
CREATE VIEW [System.Storage.Store].[Master!Extension] AS
  SELECT ... FROM [System.Storage.Store].[Table!Item]
CREATE VIEW [System.Storage.Store].[Master!Relationship] AS
  SELECT ... FROM [System.Storage.Store].[Table!Item]
```

- Each Item, Extension and Relationship type has a type specific view

```
CREATE VIEW [System.Storage.Contact.Store].Contact AS
  SELECT
    Item,
    Item.ItemId AS ItemId, ...
    Item.FirstName AS FirstName, ...
  FROM (
    SELECT TREAT( Item AS [...].Contact) as Item
    FROM System.Storage.Store.[Table!Item]
    WHERE Item IS OF ([...].Contact)
  ) AS Contact
```



# Type Search Views



# Type Indexing

- WinFS supports indexing of properties of Item, Extension or Relationship
  - Index on scalar properties Ex: Contact.BirthAddress.Zip
  - Index on collection properties Ex: Contact.Addresses[] (City, Zip)
- Indexes are supported using SQL indexed views
- Each index defines a unique index view definition which projects the required columns and filters based on type
- SQL supports fast matching of indexed views including
  - column coverage
  - type inference (index on parent types can be used to match sub-type search)





# Type Indexing

## Scalar Example

### Index on Contact(BirthAddress.Zip)

```
CREATE VIEW [System.Storage.Contact.Store].[IVCtx]
WITH SCHEMA_BINDING AS
SELECT ItemId, _SdId, _Item.HomeAddress.Zip AS Zip,
FROM (
    SELECT ItemId, _SdId, TREAT(_Item AS [...].Contact) AS _Item
    FROM [System.Storage.Store].[Table!Item] I
    WHERE _Item IS OF ([...].Contact)
) AS c

CREATE UNIQUE CLUSTERED INDEX [IVCtxCI]
ON [System.Storage.Contact.Store].[IVCtx] (Zip, ItemId)
INCLUDE (_SdId)
```



# Item Domains

- The item domain of an item defines the closure of items reachable by following holding or embedding links from the item
- Item domains are used to provide search scoping support
  - Implicitly based on the connection point  
\\machine\winfsstore\path\...
  - Explicitly using rowset functions ItemsInDomain(<itemId>)
- Item domains introduce performance requirements around path closure computation





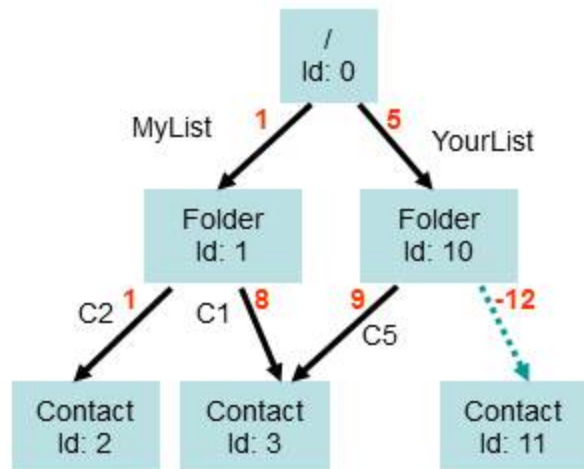
# Item Domains

- The item domain closure in the system is precomputed using a single paths table
- Each path to an item (in the DAG) is represented by a single row in the table
- Each path value is encoded using a packed binary blob called a "PathHandle"

```
CREATE TABLE [System.Storage.Store].[Table!Path] (  
    [System.Storage.Store].ItemId ItemId NOT NULL,  
    [System.Storage.Store].PathHandle PathHandle NOT NULL,  
    [System.Storage.Store].PathName Name NULL, ... )
```



# Item Domains



WinFS Paths Table

ItemID	PathHandle	Logical Path
0	\	\
1	\1	\MyList
2	\1\1	\MyList\C2
3	\1\8	\MyList\C1
10	\5	\YourList
11	\5\12	No path (embedded)
3	\5\9	\YourList\C5

.....> Embedding

————> Holding

- Each relationship is assigned an ordinal
- Embedded items are assigned negative ordinals
- Each path handle is computed by packing the ordinal values from the root to each path



# Item Domains

- The descendant limit of a path handle is computable using the function GetDescendantLimit
- A table function ItemsInDomain is defined using the paths table
- This function is applied to all WinFS type views

```
CREATE FUNCTION System.Storage.Store.ItemsInDomain
@pathHandle [System.Storage.Store].PathHandle
RETURNS TABLE AS (
    SELECT ItemId, ...
    FROM [System.Storage.Store].[Table!Path] p
    WHERE p.PathHandle
        BETWEEN @pathHandle AND DescendentLimit(@pathHandle)
)
```



# Item Security

- WinFS provides an ACL security model which implements a superset of the full NTFS security model
- ACL inheritance is propagated along holding relationships
- Each item has an “effective” permission set
- Each security zone is defined by a single column SdId which is added to all storage tables
- WinFS API support for item level security as well as Win32 and Marta integration
- Filtering in views and ItemsInDomain based on security



# Item Security

- All read operations (query) are filtered to ensure rows returned are readable by the end-user
- This is done by adding a security check predicate on all search views
- This predicate performs a win32 CheckAccess call on the underlying ACL

```
CREATE VIEW [System.Storage.Contact.Store].Contact AS
SELECT
    c._Item,
    c._Item.ItemId AS ItemId, ...
    c._Item.FirstName AS FirstName, ...
FROM (
    SELECT TREAT( Item AS [...].Contact) as _Item, _SdId
    FROM System.Storage.Store.[Table!Item]
    WHERE Item IS OF ([...].Contact)
) AS c
WHERE c._Item.ItemId IN ( ... ItemsInDomain(@@item_domain) )
    AND has_security_by_sd(_SdId) ...
```





# Update API – Manipulating Objects

- The namespace System.Storage.Store provides a core set of procedures for manipulating instances of Item, Extension and Relationship
- All procedures provided in terms of changes to the data model. CreateItem, UpdateItem, DeleteExtension
- Example:

```
-- Remove an extension  
EXEC [System.Storage.Store].DeleteExtension @itemId, @extId, ...  
  
-- Create an Item  
EXEC [System.Storage.Store].CreateItem @relationship, @item, ...
```

Creating an item  
requires a relationship





# Update API – Updating Objects

- In addition to create and delete operations, the Base API provides a granular update mechanism
- This API is utilized directly by the WinFS API
- Clients build a granular change definition (tree)
- This tree is then compiled to return a handle and parameter list
- The handle and parameters can then be used in a call to UpdateItem, UpdateExtension or UpdateRelationship
- UpdateItem et al then produce a batch of granular DML which is used to update the objects in-place



# Update API – Updating Objects

- Base uses granular SQL DML to apply changes to the underlying objects
- Example SQL batch:

```
UPDATE [System.Storage.Store].[Table!Item]
SET TREAT(_Item AS [...].Contact).(
    FirstName = TREAT(@cv[0] AS StringVal).Value,
    TREAT(Address AS [...].AddressUk).PostCode
    = TREAT(@cv[1] AS StringVal).Value,
    ...)
WHERE ItemId = @itemId AND ...

IF @@ERROR != 0 THEN BEGIN
    UPDATE ...
END
```

- Multiple queries may be required when elements are added, removed and updated to the same collection property



# File Support

- Applications can use Win32 APIs to create files in WinFS
- Files can have a metadata handler associated with them by their filename extension
  - If the file content is associated with a handler, the file will be represented as an item of a corresponding WinFS type (ex: Image, Document, AudioTrack, etc. )
  - If the file content is not associated, the file is represented as an item of type System.Storage.GenericFile
- Win32 file and folder attributes are not surfaced in the WinFS data model as schematized properties
  - Stored in separate tables
  - Accessible through Win32 APIs or views  
[System.Storage.Store].[Master!FileAttributes]



# File Support

- File storage in WinFS leverages the SQL "filestream" feature
- Filestreams are exposed as varbinary(max) columns whose binary data is stored in a real NTFS file
  - Give extremely fast streaming access (read/write)
  - WinFS is in the open and close path for file access
  - Once a file is opened, the handle is passed off to NTFS
- File stream data and attributes are stored in the File attributes table

```
CREATE TABLE [System....Store].[Table!FileAttributes] (  
    ItemId [...] .ItemId NOT NULL,  
    IsSparse BIT NOT NULL DEFAULT(0),  
    IsSystem BIT NOT NULL DEFAULT(0),  
    ...  
    Stream VARBINARY(MAX) FILESTREAM,  
    ...  
)
```





# WinFS API and Store Integration

## Query Example

- Client creates ItemContext to WinFS Store
- Submits find query using Opath
- Object faulted into item context for manipulation

```
using (ItemContext ic = new ItemContext())
{
    ic.Open();
    Contact c = ic.FindItem(
        typeof(System.Storage.Contact.Person),
        "DisplayName == 'Fred Bloggs'");
    ...
}
```

- What happened?



# WinFS API and Store Integration

## Query Example

- Client creates ItemContext to WinFS Store
  - SQL Client opens connection
- Submits find query using Opath
- Opath query compiled into TSQL and submitted over SQLClient

```
Contact c = (Contact) ic.FindItem(typeof(Contact),  
    "DisplayName = 'Fred Bloggs'");
```

```
SELECT _Item, ..., ctInfo.UpdateTS as Version  
FROM [System.Storage.Contact.Store].Contact c  
WHERE c.DisplayName = 'Fred Bloggs'
```

- Store object is returned along with change version number
- API creates a client object and stores in item context





# WinFS API and Store Integration

## Update Example

- Client retrieves object using opath search
- Object available for manipulation in item context
- Client mutates object
- Client saves all item context changes

```
using (ItemContext ic = new ItemContext())
{
    ic.Open();
    Contact c = (Contact) ic.FindItem(
        typeof(System.Storage.Contact.Person),
        "DisplayName == 'Fred Bloggs'");
    c.DisplayName = 'Fredrick Bloggs';
    c.BirthDate = '01/04/1982';
    ic.Update();
}
```

- What happened?



# WinFS API and Store Integration

## Update Example

- Client mutates object
  - Underlying API object setters track object changes
  - API builds ChangeDefinition for updated objects
- Each changed object is enumerated
  - If updated
    - The corresponding change definition is compiled
    - A call is made to the API UpdateItem, UpdateExtension, ...
    - The object timestamp is passed in to ensure correct optimistic concurrency control
  - If deleted – call DeleteRelationship or DeleteExtension
  - If created – call CreateItem, CreateExtension or CreateRelationship
- After each change is submitted to the Store, the item context is updated with the new object version



# Recap

- What we've covered
  - WinFS Storage architecture
  - Set of SQL/OR services needed by WinFS
  - WinFS logical data model mapping and physical storage mapping
  - Underlying WinFS service implementation to support indexes, change tracking, notifications, etc
  - API interaction and walkthrough



# A Note on Performance

- Significant performance issues with PDC WinFS architecture
  - PDC was a prototype
- Major implementation of underlying system just completed in December
  - Lots of room for performance work
- Q1 '04 major push on performance
  - Code work (in LAB06 in March)
  - “Early look” at how incoming work helps performance
  - Resulted in confidence that performance will be there



# WinFS Links and Aliases

- Core WinFS Site <http://winfs>
  - Links to presentations and team sub-sites
- Email aliases
  - WinFS Questions – discussions of all WinFS related issues
  - WinFS API Discussion – discussions for WinFS API users





Q/A ?

